

Introduction to Python

assignment and strings

```
a = 'a string' # <- pound char introduces a comment  
a = "a string" # ' and " chars have same functionality
```

multiline strings - use three ' or " characters

```
a = '''a  
multiline  
string'''
```

C-style string formatting - uses the % operator

```
s = "a float: %5.2f    an integer: %d" % (3.14159, 42)  
print s  
a float: 3.14    an integer: 42
```

lists and tuples

```
l = [1,2,3]          #create a list  
a = l[1]            #indexed from 0 (l = 2)  
l[2] = 42          # l is now [1,2,42]  
t = (1,2,3)         #create a tuple (read-only list)  
a = t[1]            # a = 2  
t[2] = 42          # ERROR!
```

Introduction to Python

calling functions

```
bigger = max(4,5) # max is a built-in function
```

defining functions - whitespace scoping

```
def sum(a,b):  
    "return the sum of a and b"    # comment string  
    retVal = a+b                  # note indentation  
    return retVal
```

```
print sum(42,1)                      #un-indented line: not in function
```

43

loops - the for statement

```
for cnt in range(0,3):  
    cnt += 10  
    print cnt
```

10

11

12

Introduction to Python

Python is modular

most functions live in separate namespaces called modules

The import statement - loading modules

```
import sys      #import module sys  
sys.exit(0)    #call the function exit in module sys
```

or:

```
from sys import exit #import exit function from sys into current scope  
exit(0)            #don't need to prepend sys.
```

Introduction to Python

In Python pretty much everything is an object.

Objects: calling methods

```
file = open("filename")      #open is built-in function returning an object
contents = file.read()       #read is a method of this object
                             # returns a string containing file contents
dir(file)                   # list all methods of file
[ '__class__', '__delattr__', '__doc__', '__getattribute__',
  '__hash__', '__init__', '__iter__', '__new__', '__reduce__',
  '__repr__', '__setattr__', '__str__', 'close', 'closed', 'fileno',
  'flush', 'isatty', 'mode', 'name', 'read', 'readinto', 'readline',
  'readlines', 'seek', 'softspace', 'tell', 'truncate', 'write',
  'writelines', 'xreadlines']
```

Introduction to Python

interactive help functionality: `dir()` is your friend!

```
import sys
dir(sys)      #lists names in module sys
dir()         # list names in current (global) namespace
dir(1)        # list of methods of an integer object
```

the help function

```
import ivm
help( ivm )           #help on the ivm module
help(open)            # help on the built-in function open
```

Accessing Xplor-NIH's Python interpreter

from the command-line: use the -py flag

```
% xplor -py
XPLOR-NIH version 2.9.2
```

C.D. Schwieters, J.J. Kuszewski, based on X-PLOR 3.851 by A.T. Brunger
N. Tjandra, and G.M. Clore
J. Magn. Res., 160, 66-74 (2003). <http://nmr.cit.nih.gov/xplor-nih>

```
python>
```

from XPLOR: PYTHon command

```
X-PLOR>python           !NOTE: can't be used inside an XPLOR loop!
python> print 'hello world!'
hello world!
python> python_end()
X-PLOR>
```

for a single line: CPYThon command

```
X-PLOR>cpython "print 'hello world!'"    !can be used in a loop
hello world!
X-PLOR>
```

using XPLOR, TCL from Python

to call the XPLOR interpreter from Python

```
xplor.command('''struct @1gb1.psf end  
coor @1gb1.pdb'''')
```

xplor is a built-in module - no need to import it

to call the TCL interpreter from Python

```
from tclInterp import TCLInterp          #import function  
tcl = TCLInterp()                      #create TCLInterp object  
tcl.command('xplorSim setRandomSeed 778') #initialize random seed
```

Atom Selections in Python

use the XPLOR atom selection language.

```
from atomSel import AtomSel
sel = AtomSel('''resid 22:30 and
                (name CA or name C or name N)''')
print sel.string()          #AtomSel objs remember their selection string
resid 22:30 and
                (name CA or name C or name N)
```

AtomSel objects can be used as lists of Atom objects

```
print len(sel)           # prints number of atoms in sel
for atom in sel:         # iterate through atoms in sel
    print atom.string(), atom.pos()
```

prints a string identifying the atom, and its position.

Python in Xplor-NIH

current status: low-level functionality (similar to that of XPLOR script) implemented.

partially implemented: high-level wrapper functions which will encode default values, and hide complexity.

future: develop repository of still-higher level protocols to further simplify structure determination.

Using potential terms in Python

available potential terms in the following modules:

1. rdcPot - dipolar coupling
2. noePot - NOE distance restraints
3. jCoupPot - 3J -coupling
4. xplorPot - use XPLOR potential terms
5. potList - a collection of potential terms

all potential objects have the following methods:

- | | |
|----------------|---|
| instanceName() | - name given by user |
| potName() | - name of potential term, e.g. "RDCPot" |
| scale() | - scale factor or weight |
| setScale(val) | - set this weight |
| calcEnergy() | - calculate and return term's energy |

residual dipolar coupling potential

Provides orientational information relative to axis fixed in molecule frame.

$$\delta_{\text{calc}} = D_a[(3u_z^2 - 1) + \frac{3}{2}R(u_x^2 - u_y^2)] ,$$

u_x, u_y, u_z - projection of bond vector onto axes of tensor describing orientation. D_a, R - measure of axial and rhombic tensor components.

rdcPot (in Python)

- tensor orientation encoded in four axis atoms
- allows Da, R to vary: values encoded using extra atoms.
- reads both SANI and DIPO XPLOR assignment tables.
- allows multiple assignments for bond-vector atoms - for averaging.
- allows ignoring sign of D_a (optional)
- can (optionally) include distance dependence: $D_a \propto 1/r^3$.

How to use the rdcPot potential

```
from rdcPotTools import *          #import all symbols from this module

RDC_addAxisAtoms()
rdcNH = create_RDCPot("NH",file='NH.tbl')

rdcNH.setDa(7.8)                  #set initial tensor properties
rdcNH.setRhombicity(0.3)
calcTensor(rdcNH)                 #use if the structure is approximately correct
```

NOTE: no need to have psf files or coordinates for axis/parameter atoms- this is automatic.

analysis, accessing potential values:

```
print rdcNH.instanceName()          # prints 'NH'
print rdcNH.potName()              # prints 'RDCPot'
print rdcNH.rms(), rdcNH.violations() # calculates and prints rms, violations
print rdcHN.Da(), rdcHN.rhombicity() # prints these tensor quantities
rdcNH.setThreshold(0)              # violation threshold
print rdcNH.showViolations()       # print out list of violated terms
print Rfactor(rdcNH)              # calculate and print a quality factor
```

RDCPot: additional details

using multiple media:

```
RDC_addAxisAtoms()  
rdcNH_2 = create_RDCPot("NH_2",file='NH_2.tbl')  
#[ set initial tensor parameters ]
```

multiple expts. single medium:

```
rdcCAHA = create_RDCPot("CAHA",resid=rdcNH.oAtom().residueNum(),  
                         file='CAHA.tbl')
```

rdcCAHA is a new potential term using the same alignment tensor as rdcNH.

Scaling convention: scale factor of non-NH terms is determined using the experimental error relative to the NH term:

```
scale_toNH(rdcCAHA,'CAHA')    #rescales relative to NH  
scale = (5/2)**2  
      # ^ inverse error in expt. measurement relative to that for NH  
rdcCAHA.setScale( scale )
```

NOE potential term

effective NOE distance (sum averaging):

$$R = \left(\sum_{ij} |q_i - q_j|^{-6} \right)^{-1/6}$$

Python potential in module noePot

- reads XPLOR-style NOE tables.
- potential object has methods to set averaging type, potential type, etc.

creating an NOEPot object:

```
from noePot import NOEPot  
noe = NOEPot('noe', open('noe_all.tbl').read() )
```

analysis:

```
print noe.rms()  
noe.setThreshold( 0.1 )          # violation threshold  
print noe.violations()          # number of violations  
print noe.showViolations()
```

J-coupling potential

$$^3J = A \cos^2(\theta + \theta^*) + B \cos(\theta + \theta^*) + C,$$

θ is appropriate torsion angle.

A , B , C and θ^* are set using the COEF statement in the j-coupling assignment table (or using object methods).

Use in Python

```
from jcoupPot import JCoupPot
Jhnha = JCoupPot('hnha',open('jna_coup.tbl').read())
```

analysis:

```
print Jhnha.rms()
print Jhnha.violations()
print Jhnha.showViolations()
```

using XPLOR potentials

Example using a Radius of Gyration (COLLapse) potential

```
import protocol
from xplorPot import XplorPot
protocol.initCollapse('resid 3:72')      #specify globular portion
rGyr = XplorPot('COLL')
xplor.command('collapse scale 0.1 end') #manipulate in XPLOR interface
```

accessing associated values

```
print rGyr.calcEnergy().energy          #term's energy
print rGyr.potName()                   #'XplorPot'
print rGyr.instanceName()              #'COLL'
```

all other access/analysis done from XPLOR interface.

Commonly used XPLOR terms: VDW, BOND, ANGL, IMPR, RAMA, HBDA, CDHI

collections of potentials - PotList

collection potential terms together:

```
from potList import PotList
pots = PotList()
pots.add(noe); pots.add(Jhnha); pots.add(rGyr)
pots.calcEnergy().energy                                # total energy
```

nested PotLists:

```
rdcs = PotList('rdcs')                                #convenient to collect like terms
rdcs.add( rdcNH ); rdcs.add( rdcNH_2 )
pots.add( rdcs )
for pot in pots:                                       #pots looks like a list
    print pot.instanceName()

noe
hnha
COLL
rdcs
```